Programare funcțională

Introducere în programarea funcțională folosind Haskell
C06

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Clase de tipuri

Exemplu: test de apartenență

Problemă. Să definim o funcție care testează dacă un element aparține unei liste.

- folosind recursivitate

```
myElem x []     = False
myElem x (y:ys) = x == y || myElem x ys
```

- folosind descrieri de liste

```
myElem x ys     = or [ x == y | y <- ys ]
```

- folosind funcții de nivel înalt

```
myElem x ys     = foldr (||) False (map (x ==) ys)
```

Funcția este polimorfică

Care este tipul funcției `myElem`?

Funcția `myElem` este polimorfică.

Definiția funcției este parametrică în tipul de date.

```
Prelude> myElem 1 [2,3,4]
False

Prelude> myElem ‘o‘ "word"
True

Prelude> myElem (1,‘o‘) [(0,‘w‘),(1,‘o‘),(2,‘r‘)]
True

Prelude> myElem "word" ["list","of","word"]
True
```

Funcția este polimorfică ... dar nu pentru orice tip

Totuși `myElem` nu funcționează pentru orice tip!

```
Prelude> myElem (+ 2) [(+ 2), sqrt]
No instance for (Eq (Double -> Double)) arising from a
    use of ‘elem‘
```

Ce se întâmplă?

```
myElem x ys = or [ x == y | y <- ys ]
```

```
Prelude> :t myElem
my_elem :: Eq a => a -> [a] -> Bool
```

Folosim relația de egalitate == care nu este definită pentru orice tip.

```
Prelude> ("ab",1) == ("ab",2)
False
Prelude> sqrt == sqrt
No instance for (Eq (Double -> Double)) ...
```

Clase de tipuri

Informal, o clasă de tipuri este o mulțime de funcții.

Puteți gândi o clasă de tipuri ca o interfață.

Puteți verifica ce conține o anumită clasă de tipuri folosind comanda :info sau :i.

Clasa Eq

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definitii implicite
```

Tipurile care aparțin clasei sunt instanțe ale clasei.

```
instance  Eq Bool  where
  False == False = True
  False == True  = False
  True  == False = False
  True  == True  = True
```

Clasa de tipuri. Constrângeri de tip

În signatura funcției `myElem` trebuie să precizăm ca tipul `a` este în clasa **`Eq`**

```
myElem :: Eq a => a -> [a] -> Bool
```

- **`Eq`** `a` se numește constrângere de tip.
- **`=>`** separă constrângerile de tip de restul signaturii.

Clasa de tipuri. Constrângeri de tip

În exemplul anterior `myElem` este definită pe liste, dar funcția poate fi mai complexă:

```
Prelude> :t myElem
myElem :: (Eq a, Foldable t) => a -> t a -> Bool
```

În această definiție `Foldable` este o altă clasă de tipuri, iar `t` este un parametru care ține locul unui *constructor de tip*!

Sistemul tipurilor in Haskell este complex!

Instanțe ale clasei Eq

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  (==) = eqInt   -- built-in

instance Eq Char where
  x == y = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y) = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == []     = True
  [] == y:ys   = False
  x:xs == []   = False
  x:xs == y:ys = (x == y) && (xs == ys)
```

Clasa Ord

Clasele pot fi extinse:

```
class  (Eq a) => Ord a  where
  (<)  ::   a -> a ->   Bool
  (<=) ::   a -> a ->   Bool
  (>)  ::   a -> a ->   Bool
  (>=) ::   a -> a ->   Bool
  -- minimum   definition: (<=)
  x < y   =    x <= y && x /= y
  x > y   =    y < x
  x >= y  =    y <= x
```

Clasa **`Ord`** este clasa tipurilor de date cu o relație de ordine.

În definiția clasei **`Ord`** s-a impus o constrângere de tip.
Orice instanță a clasei **`Ord`** trebuie să fie și instanță a clasei **`Eq`**.

Instanțe ale clasei Ord

```
instance  Ord Bool  where
    False <= False = True
    False <= True  = True
    True  <= False = False
    True  <= True  = True

instance (Ord a, Ord b) => Ord (a,b) where
  (x,y) <= (x',y') = x < x' || (x == x' && y <= y')
  -- ordinea lexicografica

instance  Ord a => Ord [a]  where
    []     <= ys     = True
    (x:xs) <= []     = False
    (x:xs) <= (y:ys) = x < y || (x == y && xs <= ys)
```

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie aibă o funcție care să indice modul de afișare:

```
class  Visible a  where
    toString :: a -> String
```

Putem face instanțieri astfel:

```
instance  Visible Char where
    toString c = [c]
```

Clasele Eq și Ord sunt predefinite.

Clasa Visible este definită de noi.

Există o clasă predefinită care are același rol: clasa **Show**.

Clasa Show

```
class  Show a  where
  show :: a -> String    -- analogul lui "toString"

instance  Show Bool  where
  show False      = "False"
  show True       = "True"

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

instance Show a => Show [a] where
  show []     = "[]"
  show (x:xs) = "[" ++ showSep x xs ++ "]"
    where
      showSep x []     = show x
      showSep x (y:ys) = show x ++ "," ++ showSep y ys
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)   :: a -> a -> a
  negate        :: a -> a
  ...
  fromInteger   :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       = fromInteger 0 - x


class (Num a) => Fractional a where
  (/)           :: a -> a -> a
  recip         :: a -> a
  fromRational  :: Rational -> a
  ...
  -- minimum definition: (/), fromRational
  recip x        = 1/x
```

Derivare automata pentru tipuri algebrice

Să presupunem că avem tipurile de date:

```
data Season = Spring | Summer | Autumn | Winter
```

```
data Point a b = Pt a b
```

Cum putem să le facem instanțe ale claselor `Eq`, `Ord`, `Show`?

Derivare automata pentru tipuri algebrice

Putem să le facem explicit sau să folosim derivarea automată.

```
data Season = Spring | Summer | Autumn | Winter
            deriving (Eq, Ord, Show)

data Point a b = Pt a b
               deriving (Eq, Ord, Show)
```

Atenție! Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automata vs Instanțiere explictă

Instanțierea prin derivare automată:

```
data Point a b = Pt a b
                 deriving Eq
```

Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where
       (==) (Pt x1 y1) (Pt x2 y2) = (x1 == x2)
```

Derivare automata pentru tipuri algebrice

```
data Point a b = Pt a b
               deriving (Eq, Ord, Show)
```

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
Prelude> Pt 2 3 < Pt 5 6
True

Prelude> Pt 2 "b" < Pt 2 "a"
False

Prelude> Pt (+2) 3 < Pt (+5) 6
No instance for (Ord (Integer -> Integer)) arising from
    a use of ‘<’
```

Quiz Time!

https://tinyurl.com/PF-C06-Quiz1

Definirea claselor

```haskell
class Size a where
  size :: a -> Int

instance Size Int where
  size x = abs x

instance Size [a] where
  size xs  = length xs
```

Definirea claselor

```
Prelude> :t size
size :: Size a => a -> Int

Prelude> size [True,False]
2

Prelude> sizeBoth a b = [size a, size b]

Prelude> :t sizeBoth
sizeBoth :: (Size a1, Size a2) => a1 -> a2 -> [Int]
```

Definirea claselor

O clasă poate conține mai multe funcții și chiar constante.

De exemplu, putem avea aceasta versiune a clasei Size:

```
class Size a where
  empty :: a
  size :: a -> Int
  sameSize :: a -> a -> Bool

instance Size (Maybe a) where
  empty = Nothing

  size Nothing = 0
  size (Just a) = 1

  sameSize x y = size x == size y
```

Definirea claselor

```haskell
class Size a where
  empty :: a
  size :: a -> Int
  sameSize :: a -> a -> Bool

instance Size [a] where
  empty = []
  size xs = length xs
  sameSize x y = size x == size y
```

Implementări implicite

```haskell
class Example a where
  example :: a        -- exemplul de baza pentru tipul a
  examples :: [a]     -- o lista scurte de exemple
  examples = [example]   -- implicit

instance Example Int where
  example = 1
  examples = [0,1,2]

instance Example Bool where
  example = True
```

Implementări implicite

```
Prelude> example :: Bool
True

Prelude> example :: Int
1

Prelude> examples :: [Bool]
[True]

Prelude> examples :: [Int]
[0,1,2]
```

Mini Test!

https://tinyurl.com/PF-MiniTest

Pe săptămâna viitoare!